Université Côte d'Azur — L3 Info, L3 MI, L3 ST
Automates & Langages — 2023–24

TP n°2

Reconnaissance de motifs

Présentation

La reconnaissance d'un motif dans un texte est une opération des plus courantes en informatique. En anglais cela s'appelle un problème de *pattern-matching* sur les chaînes ou plus simplement un problème de *string-matching*. Quels algorithmes de recherche de motif se cachent sous les commandes `grep` et `find` d'UNIX, `C-s` d'EMACS, *Rechercher* ... de votre traitement de texte préféré ?

Les algorithmes les plus célèbres et les plus fréquemment implantés portent le nom de leurs inventeurs : Knuth-Morris-Pratt (1977), Boyer-Moore (1974), Rabin-Karp (1987). Ils sont bien plus efficaces que l'algorithme de recherche naïf et sont encore optimisés à la programmation. Actuellement, les meilleurs algorithmes ont des complexités *sous-linéaires* c'est-à-dire qu'ils ne prennent pas connaissance de toutes les lettres du texte pour trouver un motif.

Mise en route

Aujourd'hui, on se propose d'implanter en PYTHON un algorithme de recherche naïf puis l'algorithme de Knuth-Morris-Pratt. Il faut garder à l'esprit que rechercher la première occurence d'un motif dans un texte revient à faire *glisser* de gauche à droite le motif sur le texte, jusqu'à trouver une éventuelle coïncidence.

Vous trouverez sous MOODLE de la documentation en français sur le jeu de caractères latin UNICODE, la trame d'un programme PYTHON contenant déjà la gestion entrées/sorties et des fichiers-texte pour les tests. Passons à l'écriture des fonctions de recherche.

Exercice 1 : algorithme de recherche naïf

1. Récupérez tous les fichiers du jour sous MOODLE dont le fichier `02tp.py` contenant une trame pour le TP 2.
2. Concevez un algorithme de recherche naïf prenant en entrée un motif et un texte et retournant soit la position de la première occurrence du motif dans le texte, soit -1 si le motif est absent. Implantez-le dans la fonction `recherche()`.
3. Testez votre programme sur les fichiers-texte fournis ou sur tout autre fichier.
4. Evaluez la complexité de votre algorithme dans le pire des cas après avoir identifié ce dernier. En pratique, si m est la longueur du motif et n celle du texte, la complexité moyenne est seulement en $\mathcal{O}(m+n)$.

L'algorithme de Knuth-Morris-Pratt

Cet algorithme, connu sous le nom de KMP, constitue une amélioration de l'algorithme de recherche naïf comparant un texte et un motif de la gauche vers la droite. Dès que texte et motif ne coïncidaient plus, l'algorithme naïf relançait la recherche du motif un cran plus loin. L'algorithme KMP propose de faire glisser le motif vers la droite aussi efficacement que possible.

Quand la comparaison échoue entre la lettre `motif[i]` et la lettre `texte[j]`, on sait que le préfixe `motif[:i]` est présent à la fin de la partie déjà lue du texte. On réalise alors que les possibilités pour faire glisser le motif plus efficacement vers la droite ne dépendent que du motif !

C'est l'idée de l'algorithme de KMP : le motif va faire l'objet d'un pré-traitement. Il prendra la forme d'une table de la longueur du motif. En cas d'échec ultérieur dans la comparaison de la lettre `motif[i]` avec une lettre du texte, on va consulter la table qui indique dans `table[i]` à partir d'où recommencer la recherche du motif.

La table retournée va stocker dans `table[i]` la longueur du plus long suffixe de `motif[:i]` qui est aussi préfixe du motif. On peut en déduire le meilleur décalage à droite à opérer lors du glissement du motif sur le texte sans qu'aucune occurence du motif ne soit ratée. Finalement, le pré-traitement du motif exploite les éventuelles redondances contenues dans le motif.

In fine, quand la coïncidence avec le texte a échoué à la lettre `motif[i]`, l'élément `table[i]` procure l'indice de la lettre du motif à partir de laquelle on peut reprendre les comparaisons avec le texte. Le motif glisse ainsi sur la droite sans que l'on reconsidère les lettres du texte avant la dernière lue.

Exercice 2 : pré-traitement du motif

1. On appelle *bord* d'un mot un préfixe strict de ce mot qui en est aussi un suffixe. Ici, on cherche donc à déterminer la longueur du plus long bord de `motif[:i]`. Assurez-vous que vous avez compris en vous aidant de l'exemple suivant (l'élément `table[0]` est dépourvu de sens) :

   Exemple : Le prétraitement du motif `'blalablalablabla'` donne la table :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
−1	0	0	0	0	0	1	2	3	4	5	6	7	8	1	2

   En effet, si la comparaison échoue sur la lettre `motif[8]='l'`, on cherche le plus long bord c'est-à-dire le plus long suffixe de `motif[:8]='blalabla'` qui soit aussi préfixe du motif. Ici, le bord le plus long est clairement `'bla'` et donc `table[8]` prend la valeur 3. Ce qui s'interprète ainsi : le texte lu termine par les 3 premières lettres du motif, donc on reprend la recherche directement sur sa quatrième lettre : `motif[3]`.

2. A votre tour, essayez de trouver *à la main* la table relative au motif `'murmure'` par exemple.

3. Comprenez et testez la fonction `table()` qui sert de pré-traitement du motif à l'algorithme de recherche KMP.

Exercice 3 : un début d'implantation

1. Ecrivez la fonction `rechercheMP()` qui prend en entrée un texte et un motif et qui retourne la position de la première occurrence du motif dans le texte ou -1 si le motif ne figure pas dans le texte. La recherche du motif est similaire à celle de l'algorithme naïf à ceci près qu'elle va consulter `table(motif)`. Elle va ainsi repositionner le motif sur la ou les dernière(s) lettres connue(s) du texte.
2. Comprenez le contenu de la fonction `RechercheMP()` puis utilisez-la pour tester la recherche sur des fichiers de taille conséquente.

Exercice 4 : finalisation de l'implantation

1. Pour l'instant, on a obtenu l'algorithme MP seulement. Poussez la réflexion un peu plus loin et trouvez où on peut encore l'améliorer. Vous obtiendrez alors la fonction `rechercheKMP()` sur laquelle se fonde l'algorithme KMP. Testez-la.
2. *(facultatif)* Adaptez l'algorithme afin de rechercher toutes les occurences du motif dans le texte et indiquer pour chacune sa position.

L'algorithme de Aho-Corasick

L'algorithme de Aho-Corasick date de 1975 et est implanté dans `grep` d'UNIX. Pour rechercher un motif dans un texte, il utilise tout simplement un automate fini. Vous savez construire un automate minimal pour reconnaître un motif donné. Ce que vous ne soupçonnez pas, c'est qu'un tel automate fini peut être construit inductivement sur la longueur du motif à rechercher. Ainsi, cet algorithme ne nécessite pas de prétraitement du motif : il en devient plus adapté à la recherche incrémentale (quand la recherche démarre avant même la donnée de la totalité du motif).

L'algorithme est en deux temps. Tout d'abord, il faut construire l'automate fini pour un `motif` donné, ensuite, il faut le faire fonctionner sur un `texte` donné. Nous n'aurons peut-être pas le temps de l'implanter en totalité ... mais il serait déjà bien de savoir construire l'automate fini.

Exercice 5 : construction de l'automate

1. Voici sous forme algorithmique la méthode pour construire l'automate minimal reconnaissant un `motif` sur un alphabet A donné.

```
fonction construireAutomate (motif) : automate
     créer un état init
     terminal := init
     pour tout b dans A faire
          delta  (init, b) :=  init
     pour a de première à dernière lettre du motif faire
          temp := delta (terminal,  a)
          delta (terminal,  a) := nouvel état x
          pour tout b de A faire
               delta (x,  b) := delta (temp, b)
          terminal := x
     end
     return automate
```

   Considérons l'alphabet $A = \{0, 2, 3\}$. Fabriquez à la main et en suivant cette méthode l'automate fini reconnaissant la présence du motif `2023` dans un texte.

2. Programmez cette fonction `construireAutomate()` en ayant soin de choisir une bonne représentation pour l'automate.

3. Il ne reste plus qu'à compléter votre programme pour qu'il mime le fonctionnement d'un automate sur un texte passé en paramètre.